Common.Platform.AsyncFallback — Универсальный helper для таймаутов асинхронных операций

Автор: Art Yg
Версия: 1.0

`AsyncFallback` — это платформенный helper, который решает одну практическую проблему: **как безопасно обработать ситуацию “мы отправили async-запрос, но ответ может не прийти вовремя или вообще не прийти”**.

Он нужен для любых операций, где есть “PHASE 1 → отправили запрос” и “PHASE 2 → пришёл callback”:

- LLM (Remote LLM Query)
- ImageGen (генерация изображений)
- STT / Voice Transcription
- любые RemoteApiCall (внешние API, webhook processor, интеграции)

`AsyncFallback` **не является AI-модулем**. Это оркестрация платформенного уровня.

Зачем он существует

В реальном мире async-вызовы ломаются не потому что “код плохой”, а потому что:

- провайдер завис / долго отвечает
- callback потерялся на транспорте
- webhook processor упал
- ответ пришёл, но неполный (нарушение контракта)
- пользователь продолжает писать, пока операция ещё не завершилась

Без таймаута сценарий может **залипнуть**: пользователь пишет, а система “ждёт” бесконечно.

`AsyncFallback` решает это через стандартный механизм Metabot Scheduler:

1. **Планирует job**: “через N секунд выполнить fallback-script”
2. **Отменяет job**, когда callback успешно пришёл
3. Позволяет фиксировать **причину ошибки** и **детали**, чтобы дальше можно было принять решение (редирект, retry, сообщение пользователю)

Основные принципы

- **Platform-layer** — не привязан к AI, применяется везде
- **Namespace-based** — несколько параллельных async-операций не конфликтуют
- **Opt-in** — если timeout не задан, ничего не планируется
- **Side-effect only** — не вмешивается в бизнес-логику, только планирует/снимает job и пишет маркеры
- **Debug-friendly** — сохраняет конфиг и last_reason/last_details в lead (по namespace)

Минимальные требования

Чтобы использовать `AsyncFallback`, нужно:

1. Иметь доступ к **планировщику** Metabot:
   - `bot.scheduleJob({ lead_id, script_code, run_after_sec })`
   - `bot.clearJobsByScriptCode(script_code, lead_id)`
2. Иметь `leadId` (или возможность вычислить lead_id)

`AsyncFallback` не требует базы данных, таблиц или дополнительных сервисов.

Концепция Namespace

Namespace — обязательный “scope” для идентификации конкретной операции.

Это важно, потому что на одном lead могут одновременно идти:

- ImageGen (ожидаем картинку)

- LLMQuery (ожидаем JSON)
- STT (ожидаем текст транскрипции)

Если бы мы хранили “timeout.script” без namespace — они бы перетирали друг друга.

Пример namespace:

- `orion_image_reflection`
- `llm_actor_profile_extract`
- `voice_transcription_q1`

Конфигурация

`AsyncFallback` конфигурируется на вызове:

- `namespace` — обязательно
- `timeout` — опционально
- `error` — опционально
- `storageRoot` — почти всегда не трогаем

Поля конфигурации

- `timeout.seconds` — через сколько секунд считать операцию “просроченной”
- `timeout.script` — какой скрипт выполнить по истечении таймаута
- `error.flagAttr` — атрибут флага ошибки на lead (например `true/false`)
- `error.reasonAttr` — атрибут причины ошибки (строка)

Как использовать

PHASE 1 — отправка async-запроса (isFirstImmediateCall)

1. Конфигурируем fallback
2. Планируем таймаут
3. Отправляем remote request
4. Возвращаем `false` (ждём callback)

```
const AsyncFallback = require("Common.Platform.AsyncFallback");

const fb = AsyncFallback.configure({
  lead,
  namespace: "orion_image_reflection",
  timeout: { seconds: 120, script: "Orion_Image_Timeout" },
  error: { flagAttr: "orion_image_error", reasonAttr: "orion_image_error_reason" }
});

fb.schedule();

// ... RemoteApiCall.send(..., asyncResponse: true)
return false;
```

PHASE 2 — пришёл callback

1. Снимаем таймаут (если он был)
2. Валидируем результат
3. Если контракт нарушен — `fail(reason, details)`
4. Дальше ты решаешь: редиректить в error-script или вернуться “в ту же точку”

```
const AsyncFallback = require("Common.Platform.AsyncFallback");

const fb = AsyncFallback.configure({
  lead,
  namespace: "orion_image_reflection"
});

fb.unschedule();

// если ответ плохой (например нет url при requireUrl=true)
fb.fail("url_missing", { requireUrl: true });

// твоя стратегия выхода:
return bot.run({ script_code: "Orion_Image_Error" });
```

Методы

`AsyncFallback.configure(params) → instance`

Создаёт instance и **сохраняет конфиг** в lead под namespace.
Главная точка входа. Используй и в PHASE 1, и в PHASE 2 — единообразно.

`instance.schedule() → boolean`

Планирует fallback-job, если `timeout.seconds` и `timeout.script` заданы.

Поведение:

- если timeout не задан → возвращает `false`, не считается ошибкой
- перед планированием **снимает предыдущие jobs** этого script_code для lead (защита от дублей)
- выставляет служебный флаг `active=1`

`instance.unschedule() → boolean`

Отменяет запланированный fallback-job (если он был).

Поведение:

- если script неизвестен → `false`
- снимает `active=0` даже если clearJobs вернул false (чтобы состояние не “залипало”)

`instance.fail(reason, details?) → true`

Фиксирует ошибку и причину:

- `error.flagAttr = true` (если задан)
- `error.reasonAttr = reason` (если задан)
- дополнительно пишет namespace-атрибуты:
  - `last_reason`
  - `last_details` (JSON/string)

Это **не редирект** и **не exception** — это маркер, после которого ты сам решаешь, что делать.

`instance.clear() → true`

Очищает служебные данные namespace:

- config
- script/seconds/active
- last_reason/last_details

Полезно после успешного завершения операции (опционально).

Типовой паттерн: “пользователь пишет, пока ждём”

`AsyncFallback` — про таймаут, но он закрывает важный кусок UX:

- если пользователь продолжает писать, пока async-операция ещё ждёт callback, ты показываешь “ ждите…”
- если callback так и не пришёл — fallback-job переведёт сценарий в timeout-script

Обычно это делается так:

- **проверяешь** `payload.is_async_response`
- если это не callback — отправляешь `processing` и возвращаешь `false`
- callback → `unschedule()`

(Эта логика живёт в конкретном плагине типа ImageGen/LLMQuery, а AsyncFallback даёт им общий таймаутный механизм.)

Когда использовать

- есть async callback и риск “зависнуть”
- важно гарантировать, что сценарий не будет ждать бесконечно
- нужен единый механизм таймаута для разных компонентов

- хочешь стандартизировать “timeout-script” как часть контракта компонента

Когда не использовать

- операция строго синхронная
- у операции нет понятного “deadline” (таймаут не имеет смысла)
- ты уже используешь другой механизм оркестрации таймаутов, и второй будет конфликтовать

Практические замечания, чтобы не словить редкий ад

Поздний callback после таймаута

Может случиться: таймаут-скрипт уже отработал, а потом всё же прилетел success-callback. `AsyncFallback` снимает job на callback, но **если job уже выполнился**, снять уже нечего.

Правильный паттерн на уровне компонента:

- timeout-script ставит флаг `*_timed_out = true`
- callback-обработчик проверяет флаг и **игнорирует поздний успех** (или делает компенсацию)

Это не обязанность `AsyncFallback`, потому что стратегия зависит от бизнес-логики.

Итог

`Common.Platform.AsyncFallback` — простой, но критически полезный слой платформенной оркестрации:

- даёт **гарантию выхода** из ожидания
- позволяет вести **несколько async-операций параллельно**
- стандартизирует **timeout и error markers**
- не превращается в “монолитный менеджер всего” — остаётся лёгким helper’ом

Версия #1
Artem Garashko создал 2026-02-01 10:47:36 UTC
Artem Garashko обновил 2026-02-01 10:50:21 UTC